Capítulo 1

Evolução das arquiteturas de software rumo à Web 3.0

Raoni Kulesza[1,2], Marcelo F. de Sousa[2,3], Matheus Lima[1,2], Claudiomar Araujo[1,2], Aguinaldo M. Filho[4]

[1]Centro de Informática da Universidade Federal da Paraíba (UFPB)

[2]Laboratório de Aplicações de Vídeo Digital (LAViD)

[3]Instituto de Educação Superior da Paraíba (IESP)

[4]Tribunal de Contas do Estado da Paraíba (TCE-PB)

{raoni, marcelo, matheus.lima, claudiomar.araujo}@lavid.ufpb.br,

amfilho@tce.pb.gov.br

Abstract

Web Systems were initially supported by a client-server architecture and three standards (URL, HTTP and HTML), and has evolved in the last two decades. Usability, scalability, maintenance, portability, robustness, security and integration with other systems are the main challenges of this software category. This tutorial presents the history and evolution of Web-based software architectures. We discuss current software architectural styles, patterns, and development platforms based on client-side (React JS) and server-side (Spring) technologies. In addition, we also discuss Web 3.0 requirements such as communication protocols, Microservices, MV browser-based frameworks, boilerplates client-side code, asynchronous programming, and integration with cloud computing infrastructures.*

Resumo

Os sistemas Web foram inicialmente suportados por uma arquitetura cliente-servidor e três padrões (URL, HTTP e HTML) e evoluíram consideravelmente nas últimas duas décadas. Usabilidade, escalabilidade, manutenção, portabilidade, robustez, segurança e integração com outros sistemas são os principais desafios desta categoria de software. Este capítulo apresenta a história e a evolução das arquiteturas de software baseadas na Web. Discutimos estilos de arquitetura de software atuais, padrões e

plataformas de desenvolvimento baseados em tecnologias do lado do cliente (mais especificamente, React JS) e do lado do servidor (mais especificamente, Spring). Além disso, também discutimos os requisitos da Web 3.0, como protocolos de comunicação, microsserviços, frameworks MV, código do lado do cliente com boilerplates, programação assíncrona e integração com infraestruturas de computação em nuvem.*

1.1. Introdução

Sistemas Web se tornaram popular por conta da ubiquidade dos navegadores Web, que permitem convenientemente instalar e manter sistemas de software num servidor sem ter que alterar o software do lado do cliente, mesmo que o acesso seja realizado por milhões de navegadores [Groef 2016]. Atualmente, Sistemas Web são utilizados para as mais variadas aplicações, como por exemplo: comércio eletrônico, acesso a conteúdo audiovisual, correio eletrônico, redes sociais, buscas, portais corporativos, etc [Fox 2018].

Sistemas Web podem ser considerados uma variante do modelo de software que utiliza a arquitetura cliente servidor, onde o navegador representa o cliente que interpreta código HTML, CSS e Javascript e se comunica com o servidor utilizando uma URL e o protocolo HTTP [Deitel 2012]. Inicialmente cada página Web era entregue para os navegadores como documentos estáticos, fazendo com que os servidores apenas recebem requisições para localização e envio de arquivos. Entretanto, tal conceito evoluiu e atualmente os servidores podem gerar a cada requisição uma página dinâmica por meio de execução de software, acesso à banco de dados ou integração com outros sistemas. Além disso, uma página Web também pode executar código já no lado do cliente. Tais características fizeram surgir várias plataformas de desenvolvimento de software (linguagens, bibliotecas, APIs, frameworks) tanto do lado do servidor, como do lado do cliente [Raible 2015]. Tais soluções são escritas principalmente utilizando as linguagens Java, C#, Python, Ruby ou Javascript e existem centenas de opções [Raible 2018].

Outro fator importante é que rapidamente vários Sistemas Web agregam muito valor na sua operação e normalmente tem uma abrangência global para o seu acesso. Por exemplo, o Facebook tem 1 bilhão de acessos todos os dias e o Netflix possui 81,5 milhões de clientes em 80 países [Fox e Hal 2018]. Tais características obrigam este tipo de sistema atender requisitos cada vez mais exigentes, como por exemplo: alta disponibilidade e desempenho, escalabilidade, segurança, múltiplos pontos de falha, recuperação de desastre, suporte a transações e integração com outros sistemas [Newman 2015]. Consequentemente, o uso do estilo arquitetural cliente-servidor evoluiu bastante nesta categoria de software e vários modelos são apresentados atualmente como solução [Burns 2018].

Este capítulo tem como objetivo apresentar exemplos atuais de plataformas de desenvolvimento de software *Web* tanto do lado do cliente (React JS,), como do servidor (*Spring*). Ademais, serão apresentadas um histórico da evolução de modelos arquiteturais de Sistemas Web, como por exemplo, 3 camadas, n camadas, RESTFul [Webber 2010] e Microserviços [Bóner 2016]. Por fim, também serão apresentadas soluções que foram desenvolvidas no Tribunal de Contas do Estado da Paraíba (TCE-PB) em parceira com o Laboratório de Aplicações de Vídeo Digital (LAVID) da Universidade Federal da Paraíba (UFPB) de modo a ilustrar o uso prático das

tecnologias e modelos arquiteturais num projeto. A principal contribuição é disseminar o histórico dos sistemas Web e entender as tecnologias e arquiteturas utilizadas hoje e tendências para o futuro.

1.2. Fundamentos de Sistemas Web

Esta seção aborda os fundamentos e conceitos básicos sobre sistemas *Web* necessários para o entendimento das próximas seções do presente capítulo. São abordados os seguintes assuntos: histórico e evolução da Web; os padrões URL e HTTP e, por fim, a evolução das linguagens HTML e *JavaScript*.

1.2.1. Histórico e Evolução da Web

A *Web* – também conhecida como WWW ou *World Wide Web* – foi criada por Tim Berners-Lee no início dos anos 90, pode ser compreendida como um sistema distribuído e fracamente acoplado para o compartilhamento de documentos. Mais precisamente, a ideia original de Tim para a *Web* era que ela fosse um espaço colaborativo em que as pessoas pudessem se comunicar através de informações compartilhadas [Berners-Lee 1996]. Contudo, com o passar do tempo, o surgimento de novas tecnologias, como a Computação em Nuvem [Patterson e Fox 2012], *mashups* [Yu et al. 2008], dentre outras, impulsionou o desenvolvimento da *Web*, que deixou de ser apenas um sistema distribuído de documentos interligados e se tornou uma plataforma para aplicativos e serviços abertos, interativos e distribuídos [Maximilien, Ranabahu e Gomadam 2008].

Benioff [2008] propôs uma taxonomia que foi adaptada por Burégio [2014] que pode ser adotada para ajudar na compreensão da transformação sofrida pela *Web* em que divide a história em três ondas: (i) *read only* (do inglês, *Web* apenas de leitura); (ii) *read/write Web* (do inglês, *Web* de leitura e escrita) e (iii) *programmable Web* (do inglês, *Web* programável). Como podem ser observadas na Figura 1.1, as chamadas "ondas" não são divididas pelo tempo necessariamente, mas pelo surgimento de novas funcionalidades e, dessa forma, elas podem se sobrepõe e coexistem em determinados períodos.

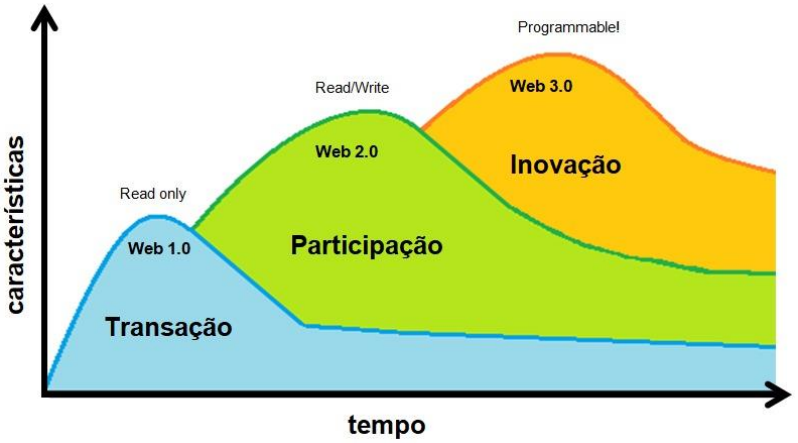

Figura 1.1. Evolução da *Web* Fonte: Adaptado de Burégio [2014]

A primeira onda da *Web*, a *read only Web*, é a chamada de *Web* 1.0 e basicamente possui aplicações capazes de prover informação em uma única direção, sendo limitadas no que diz respeito à comunicação e a interação entre os usuários.

Também são representantes da *Web* 1.0 as aplicações que permitem a realização de transações de bens e conhecimento. Dessa forma, aplicações como engenhos de busca e serviços *e-commerce* pertencem a essa primeira onda. A segunda onda da *Web*, a *read*/*write Web*, é a chamada de *Web* 2.0 e tem como principal característica a interação em comunidades, ou seja, o ponto central desta onda é a participação, colaboração e co-criação. Dessa forma, as redes sociais, *blogs*, etc, são representantes dessa segunda onda. Por último, a terceira onda da *Web*, a *programmable Web*, é a chamada de *Web* 3.0 e tem como característica a facilidade de desenvolver um sistema completo na própria *Web*, ou seja, qualquer pessoa pode criar uma nova aplicação ou serviço a partir de uma infraestrutura provida pela própria *Web*. Essa onda é impulsionada pelo advento da Computação em Nuvem. Agora, a *Web* assume o papel de uma plataforma para um ecossistema de pessoas, aplicações, serviços e até mesmo objetos físicos (*Internet of Things* – IoT).

1.2.2. Os padrões URL e HTTP

É necessário entender o funcionamento de algumas tecnologias para obter uma melhor compreensão da dinâmica existente entre os sistemas *Web* modernos. Destacam-se os seguintes conceitos fundamentais, por meio dos quais é possível desenvolver grande parte das aplicações *Web* atuais: recursos e suas representações; URIs; e ações ou verbos.

Recursos podem ser compreendidos como dados e informações – como um documento, um vídeo ou um dispositivo qualquer –, que podem ser acessados ou manipulados por meio dos sistemas baseados na *Web*. Muitos recursos do mundo real podem ser representados na *Web*, sendo necessária apenas a devida abstração de quais informações são adequados para representar estes recursos. Essa estratégia torna a *Web* uma plataforma heterogênea e acessível, pois praticamente qualquer coisa pode ser representada como recurso e disponibilizada na *Web* [Webber, Parastatidis e Robinson 2010]. A partir do momento que um recurso é publicado na Web é necessário uma forma de identificá-lo na rede, bem como de acessá-lo e manipulá-lo. Para tanto, a Web provê a URI (do inglês, *Uniform Resource Identifier*), que estabelece uma forma de identificação dos recursos por meio de um relacionamento de um-para-muitos, ou seja, uma URI identifica apenas um recurso, mas um recurso pode ser identificado por muitas URIs. Mais precisamente, um recurso como uma *Playlist*, por exemplo, pode ser representada pela linguagem de marcação HTML e interpretada pelos navegadores *Web*. De maneira similar, a *Playlist* também pode ser representada em formato XML/JSON, usualmente utilizado por outros sistemas e máquinas. A Figura 1.2 exemplifica a representação de um recurso com diversas URIs e representações.

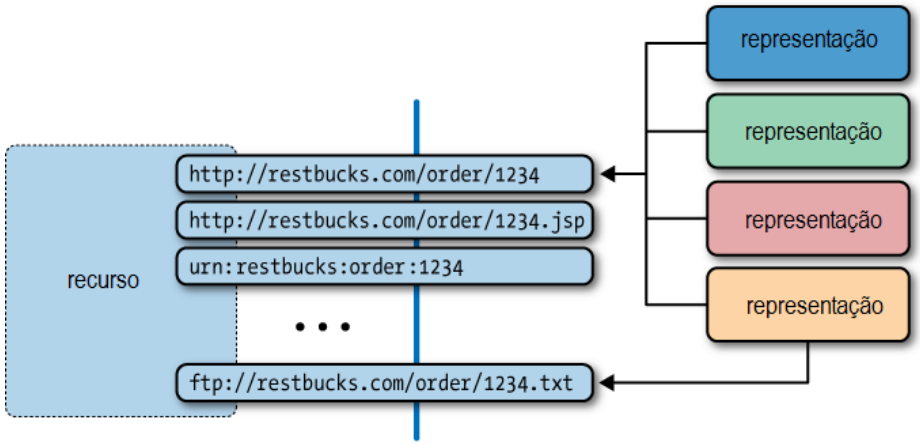

Figura 1.2. Princípios da Web [Webber, Parastatidis e Robinson 2010]

Uma URI identifica o mecanismo pelo qual um recurso pode ser acessado é geralmente referido como um URL (do inglês, *Uniform Resource Locator*). URIs HTTP são exemplos de URLs. O HTTP [RFC7231 2018] (do Inglês, *HyperText Transfer Protocol*) é um protocolo da camada de Aplicação do modelo OSI (do Inglês, *Open System Interconnection*) utilizado para transferência de dados na Internet. É por meio deste protocolo que os recursos podem ser manipulados. Para tanto, existem os chamados verbos ou ações providas pelo HTTP. A especificação original do HTTP fornece uma série de métodos de requisição responsáveis por indicar a ação a ser executada na representação de um determinado recurso. Esses métodos também são conhecidos como verbos HTTP (do inglês, HTTP *Verbs*). Os verbos HTTP utilizados para interação com os recursos *Web* são:

- **GET**: é utilizado para solicitar uma representação de um recurso específico e devem retornar apenas dados.
- **HEAD**: similar ao método GET, entretanto, não possui um corpo "*body*" contendo o recurso.
- **POST**: é utilizado para submeter uma entidade a um recurso específico, podendo causar eventualmente uma mudança no estado do recurso, ou ainda solicitando alterações do lado do servidor.
- **PUT**: substitui todas as atuais representações de seu recurso alvo pela carga de dados da requisição.
- **DELETE**: remove um recurso específico.
- **CONNECT**: estabelece um túnel para conexão com o servidor a partir do recurso alvo;
- **OPTIONS**: descreve as opções de comunicação com o recurso alvo.
- **TRACE**: executa uma chamada de *loopback* como teste durante o caminho de conexão com o recurso alvo;
- **PATCH**: aplica modificações parciais em um recurso específico.

A partir desses três conceitos fundamentais (recursos; URIs e ações) foi possível construir vários modelos de arquiteturas de software para o desenvolvimento de sistemas Web. Nas próximas duas seções serão abordadas as tecnologias de desenvolvimento do lado do cliente (seção 1.3) e servidor (seção 1.4) que apoiam a implementação desses sistemas.

1.3. Tecnologias do cliente para desenvolvimento de Sistemas Web

Com o sucesso e o aumento de seu acesso, a complexidade do conteúdo servido pela Web evoluiu, fazendo com que os conteúdos, antes apenas estáticos, tomassem forma de aplicações com capacidade de se comportar de forma similar a aplicações *desktop*. A partir dessa premissa, em 1995, a *Netscape Communications* apresentou o *JavaScript*, uma linguagem de *script* do lado do cliente que permite aos programadores melhorar a interface e interatividade do usuário com os elementos dinâmicos.

Em 2005, Jesse James Garrett tinha como objetivo a popularização de uma mudança no que o *Javascript* era capaz de fazer. Ele propôs uma abordagem na construção de aplicações Web chamada AJAX (*Asynchronous Javascript + XML*). A proposta apresentava uma mudança significativa no método tradicional das aplicações Web. No modo tradicional, interações do usuário efetuavam requisições HTTP para um servidor, que processava o pedido e retornava uma nova página HTML. A proposta de Garrett era adicionar uma camada responsável por solicitar dados ao servidor e realizar todo o processamento sem a necessidade de atualizar toda a estrutura do documento HTML que estava em exibição, tornando assim a comunicação entre cliente e servidor assíncrona [Garrett 2005]. Com o advento do AJAX, as páginas HTML tornaram-se mais amigáveis, visto que enviar os dados para o servidor para possibilitar a geração de toda a página Web deixou de ser sempre necessário.

Entretanto, a adoção do *JavaScript* não percorreu um caminho simples, principalmente, devido a competição entre os implementadores de navegadores Web que buscavam soluções específicas para os seus produtos, na maioria das vezes incompatíveis entre si. Tal contexto motivou a comunidade de desenvolvimento *JavaScriptt* a implementar bibliotecas e frameworks para mitigar esse problema, e oferecer comportamento uniforme e produtividade, como por exemplo jQuery[1]

Seguindo o movimento de sucesso do AJAX e consolidação do HTML5 e das ferramentas de produtividade para melhorar, principalmente, a implementação da interface gráfica com o usuário, foi proposto a expansão dessas facilidades para manipulação de todo o aplicativo do lado do cliente, surgindo o conceito de SPA (do Inglês, *Single Page Application*), que é um tipo de aplicação na qual carrega uma página HTML única, juntamente com seus recursos *Javascript* e CSS. Após isso, o navegador será responsável por reescrever dinamicamente à página atual em vez de carregar páginas novas inteiras de um servidor, minimizando o tráfego cliente-servidor. Com isso, o *browser* conterá mais lógica e será capaz de executar funções como renderização do HTML, validação, mudanças na interface do usuário e assim por diante [Mikowski 2013].

O *Javascript* cresceu bastante ao longo dos anos, possuindo uma comunidade bastante ativa que constantemente encontrou limitações e construiu ferramentas que supriram tais necessidades. Nos dias atuais, os desenvolvedores possuem várias alternativas modernas para criação da interface de usuário. Por exemplo: AngularJS[2], Ember[3], ReactJS[4], VueJS[5]. Na próxima seção, iremos nos aprofundar no ReactJS.

[1] https://jquery.org
[2] https://angular.io/
[3] https://www.emberjs.com/
[4] https://reactjs.org/

1.3.1. ReactJS

Ao longo da história da *Web*, várias bibliotecas *Javascript* foram desenvolvidas para tentar resolver os problemas de lidar com interfaces de usuário complexas. Entretanto, essas bibliotecas ainda mantinham a maneira clássica de separação de responsabilidades que divide o estilo (CSS), dados, estrutura (HTML) e interações dinâmicas (JavaScript).

O *ReactJS* é uma biblioteca *Javascript* para criação de interfaces, criada e mantida pelo *Facebook* [Facebook 2018]. Diferentemente de outras abordagens, ele simplifica o desenvolvimento *front-end*, pois sua principal estratégia é o Desenvolvimento Baseado em Componentes (do Inglês, *Component Driven Development*). Assim, em vez de definir um modelo único para suas interfaces, elas são divididas em pequenos componentes reutilizáveis, ou seja, o princípio é sempre buscar diminuir a complexidade por meio da separação em componentes [Mardan 2017]. A ideia é facilitar o reuso, além de promover outros benefícios como a manutenibilidade e desenvolvimento distribuído, além de integrar facilmente ao processo de desenvolvimento. Vale salientar que a criação de interfaces de usuários (UIs) componentizadas não é uma abordagem nova, porém, o *React* foi o primeiro a fazê-lo a partir do *JavaScript* puro sem uso de modelos.

O *React* não é uma estrutura *front-end Javascript* completa. Ele não estabelece uma maneira específica de desenvolver modelagem, estilo ou roteamento de dados. O *React* funciona como o "V" do modelo de arquitetura MVC (do Inglês, *Model View Controller*). Por conta disso, os desenvolvedores necessitam unir o *React* com uma biblioteca de roteamento ou modelagem. O desenvolvedor é livre para escolher quais bibliotecas utilizar, porém existe uma *React Stack* bastante adotada para auxiliar na construção de uma aplicação *front-end* completa [Mardan 2017]. Essa *stack* consiste em bibliotecas de dados e roteamento criadas para serem usadas especificamente com o *React*. Para o modelo de dados, temos, por exemplo, o *RefluxJS*[6], *Redux*[7], *Meteor*[8], *Flux*[9]. Para biblioteca de roteamento é recomendado utilizar o *React Router*[10]. E para estilização da interface do usuário é possível utilizar a coleção de componentes *React* que consomem a biblioteca do *Twitter Bootstrap*[11], o *React-Bootstrap*[12].

1.3.1.1. *Single Page Application* e ReactJS

O *React* possibilita a construção de uma SPA, embora não seja sua única forma de implementação. O código escrito em *React* pode coexistir com a marcação renderizada no servidor por algo como o ou com outras bibliotecas do lado do cliente.

Para exemplificar, assumindo que sua SPA utilize uma arquitetura do tipo MVC. O *navigator* da aplicação, funcionando como o "C" do padrão arquitetural MVC, determina quais dados buscar e qual modelo (*Model*) utilizar. Ele também realiza solicitações para obtenção de dados e preenche os *templates* (*Views*) a partir dos dados

[5] https://vuejs.org/
[6] https://github.com/reflux/refluxjs
[7] http://redux.js.org
[8] https://www.meteor.com
[9] http://facebook.github.io/flux/
[10] https://reacttraining.com/react-router/
[11] http://getbootstrap.com/
[12] https://react-bootstrap.github.io

obtidos para renderizar a interface do usuário na forma do HTML. A interface do usuário envia ações de volta ao SPA, tais como eventos do mouse, eventos de teclado, entre outros [Mardan 2017].

1.3.1.2. Virtual Dom

Um ponto que diferencia as aplicações desenvolvidas em *ReactJS* é o uso do *Virtual* DOM (VDOM), e é fundamental entendê-lo para compreender o funcionamento básico de uma aplicação desenvolvida em *ReactJS*. Trata-se de um conceito de programação na qual é criada uma representação "virtual" da interface do usuário em *JavaScript* puro, sendo mantida na memória e sincronizada com o *Document Object Model* (DOM) "real" por uma biblioteca como o *ReactDOM* [Facebook 2018].

Assim, a aplicação passa a manipular o VDOM e não o DOM diretamente. Uma das razões para a adoção do VDOM é que para que sejam realizadas as atualizações necessárias do DOM, sua estrutura pode executar diversas atualizações desnecessárias, causando assim perda de desempenho, principalmente para os casos em que a interface do usuário é complexa [Mardan 2017]. Com isso, a cada alteração no VDOM, um algoritmo primeiro calcula a diferença entre o VDOM e o DOM real e, a partir dessa análise, a biblioteca é capaz de identificar uma mudança na renderização, atualizando somente a mudança no DOM real [Chedeau 2013].

1.3.1.3 JavaScript Syntax eXtension

JSX (do Inglês, *JavaScript Syntax eXtension*) é uma extensão de sintaxe para escrever *Javascript* como se fosse XML. O JSX não é executado no navegador, mas é usado como o código-fonte para a compilação. Ele é transpilado em *Javascript* regular. Seu uso é opcional, porém é recomendado pelo *Facebook* para o desenvolvimento de aplicações em *React*. Apesar de parecer uma linguagem de modelo, o JSX possui o mesmo poder do *JavaScript* e produz elementos *React*. A Listagem 1.1 a seguir apresenta um trecho de código escrito sem o uso do JSX:

Listagem 1.1. Exemplo de Código sem o uso do JSX

```
1.const element = React.createElement("p", null, "Hello");
```

Já a Listagem 1.2 exemplifica o uso do JSX. Nela é possível observar que a sintaxe JSX ajuda a diminuir a verbosidade e facilita a criação de elementos *React*.

Listagem 1.2. Exemplo de Código com o uso do JSX

```
1.const element = <p>Hello</p>;
```

1.3.1.4. Gerenciador de pacotes

Para gerenciar todas as dependências de uma aplicação é utilizado um gerenciador de pacotes. No contexto deste trabalho, foi utilizado o gerenciador de pacotes npm[13] (*Node Package Manager*) que é distribuído com o Node.js[14], o que significa que ao realizar o

[13] https://www.npmjs.com
[14] https://nodejs.org

download do Node.js o npm é automaticamente instalado. Para verificar se a instalação acorreu adequadamente, basta executar os comandos apresentados na Listagem 1.3 no terminal:

Listagem 1.3. Comandos para verificação de instalação do node e npm

```
> node -v
> npm -v
```

1.3.1.5 *Create React App*

Create React App[15] é um *React toolchain* que constrói de forma simples e prática um *boilerplate* de uma aplicação SPA em *React*. Esta abordagem não é a única para utilizar o *React*, porém ela é recomendada para aqueles que estão iniciando no mundo do *React*, pois ela abstrai etapas de configurações importantes para uma aplicação *React* funcionar, permitindo ao programador focar apenas no código.

Para utilização do *Create React App* é necessário ter devidamente instalado em um computador a versão do *Node* >= 6 e npm >= 1.2. Para tanto, basta executar no terminal o comando descrito na Listagem 1.4 a seguir:

Listagem 1.4. Comandos para instalação do node

```
1. node install create-react-app -g
```

1.3.1.6. Primeira aplicação

Para criar a primeira aplicação *React* é necessário executar o seguinte comando no terminal, conforme a Listagem 1.5:

Listagem 1.5. Comandos para a primeira aplicação *React*

```
1. create-react-app minicurso-webmedia-react
2. cd minicurso-webmedia-react
3. npm start
```

Como já explanado, a linha 1 do código irá construir o *boilerplate* da aplicação. O *create-react-app*, além de outros arquivos de configuração, também instalará as bibliotecas *React* e *ReactDOM.* Após executá-lo, um diretório com o nome da aplicação é criado. Dentro deste diretório conterá todos os arquivos necessários para o desenvolvimento ser iniciado. Já o comando da linha 3 permite rodar a aplicação em um servidor local. O *script* "*start*" executado por intermédio do comando npm irá executar internamente o *script react-scripts start* que é responsável por configurar o ambiente de desenvolvimento e iniciar um servidor, bem como o *hot module reloading* que irá atualizar a aplicação automaticamente a cada alteração no código. Por fim, após iniciar a aplicação, ela estará acessível em http://localhost:3000/. A Figura 1.3 apresenta como estão dispostos os arquivos da estrutura inicial do projeto *ReactJS*:

[15] https://github.com/facebook/create-react-app

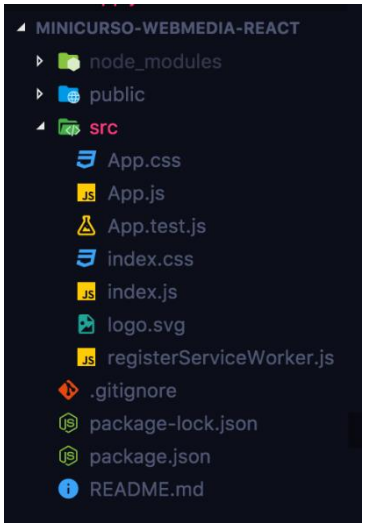

Figure 1.3. Estrutura inicial do projeto em ReactJS

O arquivo *App.css* contém algumas estilizações prontas criadas a partir do *boilerplate* da aplicação. Caso seja necessário é possível criar novas classes CSS para serem utilizadas na aplicação. O arquivo *index.js* é o ponto de entrada para a aplicação e a partir dele é chamado o arquivo *App.js*, que contém a implementação inicial de fato que é visualizada ao abrir a aplicação no servidor local. Portanto, para construir uma aplicação "Olá Mundo!" é necessário alterar o conteúdo do arquivo *App.js* conforme o código apresentado na Listagem 1.6:

Listagem 1.6. Aplicação "Olá Mundo!

```
1.import React, { Component } from 'react';
2.
3.class App extends Component {
4.    render() {
5.        return <h1>Olá mundo!</h1>;
6.    }
7.}
8.
9.export default App;
```

Vale salientar que o código apresentado na Listagem 1.6 está escrito em JSX. Além disso, ele também contém funcionalidades novas do *JavaScript* descritas no ES2015+ (*EcmaScript* 2015) ou popularmente conhecida como ES6+[16]. Por exemplo, na linha 1 é utilizada a palavra-chave *import* ao invés de ser utilizado o *require* para importar uma biblioteca externa. Na linha 2 é utilizado o conceito de *class* e na linha 9 é possível observar a utilização da palavra-chave *export*. Ao salvar este trecho de código, a aplicação irá ser atualizada automaticamente gerando o resultado que pode ser observado na Figura 1.4

[16] http://es6-features.org/

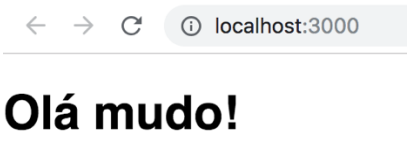

Figure 1.4. Resultado "Olá Mundo!" no navegador

1.3.1.7. Aplicação Lista de Tarefas

Para aprofundar o conhecimento sobre o *React*, esta subseção do presente capítulo demonstra a construção de uma aplicação simples baseada na ideia de listagem de tarefas a partir do mesmo projeto, previamente apresentado.

1.3.1.7.1. *Components* e *props*

Em uma aplicação *React* é necessário pensar em usar a arquitetura baseada em componentes (*Components*), que permite reutilizar o código separando a funcionalidade em partes fracamente acopladas. Por exemplo, na aplicação de lista de tarefas, ao invés de repetir o código de um item de tarefa na lista, é possível criar um componente denominando, neste caso, de "*TaskItem*" e reutilizá-lo sempre que for necessário. Adotando essa estratégia, o código torna-se escalável, legível, reutilizável e simples de mantê-lo. Essa abstração permite a reutilização de interfaces de usuário em aplicativos grandes e complexos, bem como em projetos diferentes.

As *tags* padrões do HTML (*div*, *input*, *p*, *h1*, dentro outras) podem ser utilizadas para compor as classes de componentes *React*, assim como outros componentes. Isso permite uma flexibilidade para criação de componentes robustos e potencialmente reutilizáveis. Teoricamente, os componentes são como funções *JavaScript*. É possível fornecer entradas de dados chamadas de "*props*" (que significa propriedades), e eles retornam elementos *React* que descrevem o que deve ser exibido na tela. Na linha 4 do trecho de código apresentado na Listagem 1.7, "*TaskItem*" é um componente que está recebendo como *props* um objeto *JavaScript*, descrito na linha 1, denominado "*task*". O componente "*TaskItem*" pode acessar internamente o objeto "*task*" por intermédio de suas *props*, definindo assim o que fazer com esse objeto, como por exemplo exibi-lo na tela.

Listagem 1.7. Funcionamento de *Components* e *props*

```
1.task = { title: "Estudar React" };
2.// ...
3.render() {
4.    return <TaskItem task={task} />;
5.}
6.// ...
```

É possível definir um componente de várias formas, sendo a forma mais simples por meio de uma função *Javascript*. Na Listagem 1.8 é possível observar que o objeto *props* é definido como argumento da função.

Listagem 1.8. Funcionamento de *Components* e *props*

```
1.function TaskItem(props) {
2.    render() {
3.        return <li>{props.task.title}</li>;
4.    }
5.}
```

Essa função é um componente *React* válido, pois aceita um único argumento, no caso o objeto "*props*", e retorna um elemento *React*. Componentes que são criados literalmente como funções *JavaScript* são denominados de componentes funcionais.

Dando continuidade ao assunto, outra forma de definir um componente é por meio de classes (classes ES6). Observe o trecho de código apresentado na Listagem 1.8:

Listagem 1.8. Funcionamento de *Components* e *props*

```
1.class TaskItem extends React.Component {
2.    render() {
3.        return <li>{this.props.task.title}</li>;
4.    }
5.}
```

A principal diferença entre as duas abordagens é que a segunda traz métodos adicionais, tais como métodos de ciclo de vida dos componentes provenientes da extensão "*React.Component*" (linha 1). Portanto, a decisão de uma em detrimento de outra depende apenas do que o programador deseja fazer com o seu componente.

1.3.1.7.2. Manipulando *state*

Para iniciar o desenvolvimento, será adicionado um elemento <*input*> que é responsável por receber do usuário o nome das tarefas que ele deseja adicionar em sua lista de tarefas. Para tanto, é necessário controlar o *input* para que seja possível ter acesso ao seu valor e também atualizá-lo sempre que o usuário digitar um novo caractere.

Dessa forma, esse é o momento adequado para introduzir o conceito *state*. O *state* é similar ao *props*, pois também se trata de um objeto *Javascript*, porém ele é privado e totalmente controlado pelo seu componente, ou seja, apenas o próprio componente pode alterá-lo. Suas alterações, assim como as das *props*, geram uma atualização no componente, afetando sua visualização e possíveis lógicas internas. *Props* e *states* podem possuir comportamentos semelhantes, mas eles são utilizados para finalidades diferentes.

Em HTML, elementos de formulário, como <*input*>, normalmente mantêm seu próprio estado e o atualizam com base na entrada do usuário. Em *React*, deve-se tratar o estado mutável no estado (*state*) dos componentes e atualizá-lo apenas com a função "*setState()*". Portanto, para trabalhar com esse tipo de situação deve-se manter uma única fonte e, dessa forma, é necessário combinar os dois no estado do componente. Para isso existe um *controlled component* (componente controlado), ou seja, um elemento de formulário de entrada cujo valor é controlado pelo *React*.

O código apresentado na Listagem 1.9 demonstra o acesso único ao valor contido no *input* (linha 14) que pode ser atualizado sempre que necessário. Para tanto, é utilizado o método provindo da classe *Component* chamado "*setState()*" (linha 5). Sempre que se faz necessário atualizar o estado de um componente este método é acionado, pois só assim o componente consegue saber que houve uma alteração no seu estado e que ele necessita renderizar novamente. Por exemplo, caso ocorra uma tentativa de alteração do estado de forma direta, o componente não saberá que houve atualização e consequentemente essa ela não será renderizada. Na linha 2 o objeto *state* do componente *App* é definido. Em seguida, na linha 4, o método que será responsável por acionar o *setState* é definido. O método *onChangeInputValue* é passado como valor da *props onChange* do elemento <*input*> (linha 17) e o valor do elemento <*input*> será o estado do componente, no caso, *state.inputValue* (linha 16).

Listagem 1.9. Exemplo de Código para manipulação do *state*

```
 1.class App extends Component {
 2.    state = { inputValue: '' };
 3.
 4.   onChangeInputValue = (event) => {
 5.        this.setState({ inputValue: event.target.value });
 6.    }
 7.
 8.    render() {
 9.        return (
10.            <div>
11.                <form>
12.                    <label>
13.                        Tarefa:
14.                        <input
15.                            type="text"
16.                            value={this.state.inputValue}
17.                            onChange={this.onChangeInputValue}
18.                        />
19.                    </label>
20.                </form>
21.            </div>
22.        );
23.    }
24.}
```

Vale ressaltar que O JSX nos permite que seja utilizado *props* (*value* e *onChange*) no elemento <*input*>. Ainda sobre o código da Listagem 1.9, o trecho da linha 14 a 18, que contém o elemento <*input*>, será compilada para o código a seguir da Listagem 1.10:

Listagem 1.10. Código gerado a partir do elemento <*input*>

```
1. React.createElement(
2.     'input',
3.     { value: this.state.inputValue, onChange: this.onChangeInputValue },
4.     null
5. );
```

O próximo passo pode ser observado na Listagem 1.11 em que é criada uma lista para armazenar as tarefas adicionadas pelo usuário. Assim, é adicionado mais um estado no componente denominado "*tasks*". Além disso, um *input* do tipo *submit* é criado (linha 26) para que o usuário adicione uma tarefa ao submeter o formulário.

Listagem 1.11. Código de lista para armazenamento das tarefas adicionadas pelo usuário

```
 1.   // ...
 2.    handleSubmit = (event) => {
 3.        event.preventDefault();
 4.        this.addTask();
 5.    }
 6.
 7.    addTask = () => {
 8.        const tasks = this.state.tasks;
 9.        const task = { title: this.state.inputValue };
10.        tasks.push(task);
11.        this.setState({ tasks: tasks });
12.    };
13.
14.    render() {
15.        return (
16.            <div>
17.                <form onSubmit={this.handleSubmit}>
18.                    <label>
19.                        Tarefa:
20.                        <input
21.                            type="text"
22.                            value={this.state.inputValue}
23.                            onChange={this.onChangeInputValue}
24.                        />
25.                    </label>
26.                    <input type="submit" value="Adicionar" />
27.                </form>
28.            </div>
29.        );
30.    }
```

Analisando o trecho de código da Listagem 1.11, na linha 2 é definido o método que será chamado quando o usuário inserir uma nova tarefa. Esse método é passado via

props para o elemento <*form*>. Na linha 7, o método *addTask* é responsável por adicionar uma tarefa na lista de tarefas. Vale ressaltar que ele acessa o estado do componente (linha 8) e o atualiza por intermédio do método *setState* (linha 11).

1.3.1.7.3. Exibindo lista de tarefas

Por ser possível adicionar as tarefas em uma lista, agora também é permitido exibi-las na tela do usuário. Esta atividade pode ser realizada de várias maneiras, inclusive criando um novo componente, contudo, será adotada a estratégia mais simplista. Na Listagem 1.12 pode ser observado como foi construído o método para renderizar os itens na tela e executar essa função no método "*render()*" (linha 15) do componente *App*.

Listagem 1.12. Código para exibição da lista de tarefas

```
1.    renderTasks = () => {
2.        return (
3.            <ul>
4.                {this.state.tasks.map((task, index) => (
5.                    <TaskItem
6.                        onClick={() => this.removeTask(index)}
7.                        key={index}
8.                        task={task}
9.                    />
10.               ))}
11.           </ul>
12.       );
13.   };
14.
15.   render() {
16.       return (
17.           <div>
18.               <form onSubmit={this.handleSubmit}>
19.                   <label>
20.                       Tarefa:
21.                       <input
22.                           type="text"
23.                           value={this.state.inputValue}
24.                           onChange={this.onChangeInputValue}
25.                       />
26.                   </label>
27.                   <input type="submit" value="Adicionar" />
28.               </form>
29.               {this.renderTasks()}
30.           </div>
31.       );
32.   }
```

Discutindo melhor o trecho de código da Listagem 1.12, o método "*renderTasks()*", descrito na linha 1, acessa a lista de tarefas pertencente ao estado do componente. Em *React*, transformar *arrays* em listas de elementos é similar ao método *Javascript* tradicional. No método para renderizar as tarefas, o *array* é percorrido de *tasks* utilizando a função "*map()*" (linha 4) do *Javascript*. Assim, é retornado um componente *TaskItem* (linha 5), para cada item, com suas respectivas *props* (linhas 6, 7 e 8). O componente *TaskItem*, como definido anteriormente, é um componente que retorna um elemento *<li>*.

1.3.1.7.4. Removendo itens da lista de tarefas

No trecho de código da Listagem 1.13 correspondente à implementação do componente *App*, foi definido o método *removeTask* (linha 15) que acessa o estado do componente (linha 16) e o atualiza com a nova lista de *tasks* (linha 18) após remover a tarefa solicitada pelo usuário. No método *renderTasks* (linha 21), mais especificamente na chamada do componente *TaskItem* (linhas 25 a 29), é passado por meio da *props onClick* o método *removeTask* (linha 15). Com isso o componente *TaskItem (linha 1)* acessa e executa (linha 5), ao usuário clicar no elemento, o método recebido em suas *props* que irá remover o item da lista clicado. Dessa forma a aplicação de lista de tarefas está concluída. Vale ressaltar que por motivos didáticos não foi utilizada a estilização CSS nesta aplicação. Assim, foi possível focar nos conceitos do *React*. O código fonte apresentado está disponível no GitHub[17]. Para executar a aplicação, após clonar o projeto, basta executar o comando *npm install* para baixar todas as dependências do projeto. Em seguida, digite o comando *npm start* para executar a aplicação.

Para finalizar a lista de tarefas, o usuário deve ser capaz de remover uma tarefa ao concluí-la. Existem várias maneiras de excluir um item da lista, sendo aqui desenvolvida uma abordagem simples que exclui um item por meio do clique apresentada na Listagem 1.13. Porém é necessária cautela na forma que isso será implementado na aplicação *React*. O componente *TaskItem* recebe via *props* a *task* a ser exibida, portanto ele não pode alterá-la diretamente, visto que as *tasks* pertencem ao estado do componente *App* e somente ele pode alterá-las. Para resolver este impasse, o componente *TaskItem* recebe uma função por intermédio de suas *props*, que executa (linha 5) por meio do clique do usuário excluindo o item da lista. Essa função fica localizada no componente *App* (linha 15), que tem acesso e pode alterar o seu estado.

[17] https://github.com/rkulesza/webdev

Listagem 1.13. Código para remoção de itens da lista de tarefas

```
class TaskItem extends React.Component {
  render() {
      return (
          <li
              onClick={this.props.onClick}
          >
              {this.props.task.title}
          </li>
      );
  }
}

 class App extends Component {
   // ... App component ...
   removeTask = (index) => {
     const tasks = this.state.tasks;
     tasks.splice(index, 1);
     this.setState({ tasks: tasks });
   };

   renderTasks = () => {
     return (
         <ul>
             {this.state.tasks.map((task, index) => (
                 <TaskItem
                     onClick={() => this.removeTask(index)}
                     key={index}
                     task={task}
                 />
             ))}
         </ul>
     );
   };
  // ...
 }
```

1.4. Tecnologias de servidor para desenvolvimento de Sistemas Web

Esta seção apresenta exemplos do uso de uma tecnologia de desenvolvimento do lado do servidor: Spring

1.4.1. Framework Spring

Spring fornece modelos de programação e configuração para aplicações Java EE modernas em qualquer tipo de plataforma. Seu principal foco é infraestrutura para o nível de aplicação, permitindo as equipes se concentrarem nas regras de negócios. Ele é

dividido em módulos, oferecendo suporte para desenvolvimento *Web*, *Aspect Oriented Programming* (AOP), processamento de dados, transações e integração com outras tecnologias; possui *Dependency Injection* (DI) através do seu container e suporte a testes de funcionalidades implementadas com seus módulos. A Figura 1.5 apresenta uma visão geral dos módulos que compõe o ecossistema Spring que serão detalhados nas próximas seções.

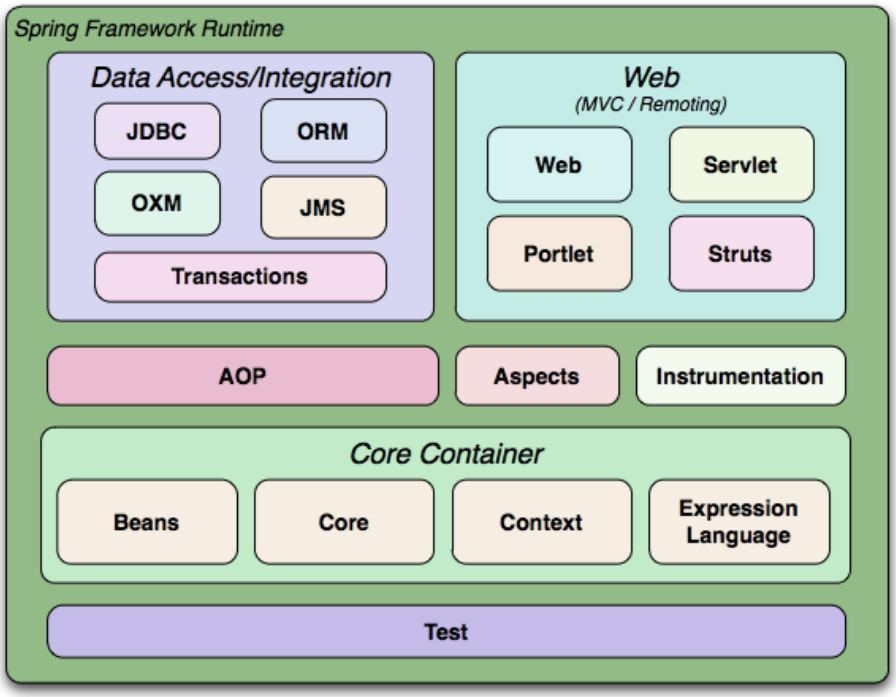

Figure 1.5 Módulos Spring Framework, Fonte: Pivotal [2018a]

1.4.1.1. Spring Core e Dependency Injection

O núcleo do Spring é um *container* que cria e gerencia *beans* automaticamente. *Beans* são instancias de objetos pertencentes ao *container Inversion of Control* (IoC). IoC é um princípio da engenharia de software que também é conhecido como *Dependency Injection* (DI). A instância de um objeto é controlada pelo *container* sendo "injetada" em outro objeto que possui dependência com o primeiro. Isso permite conexão desacoplada, aumenta a modularidade do programa e o torna mais extensível, como demonstra [Hall 2017]. Um cenário de uso real é quando o acesso a instância de um objeto é necessário em diferentes partes do programa ou quando um objeto acessa conexão entre camadas. Para realizar a injeção, ele procura por *beans* no contexto da aplicação por meio de *component scanning* e satisfaz suas dependências com *autowiring*. *Beans* possuem escopo padrão *singleton* por *container*, o que significa que é criada apenas uma instância por *container*, podendo inclusive ser configurado com outras opções de escopo. Essa implementação é um pouco diferente do padrão definido por [Gamma et al. 1996], onde o escopo é feito por classe. Uma *bean* deve ser definida em um contexto de configuração, e seu *Plain Old Java Object* (POJO) permanece o mesmo, sem invasão do *framework*.

Com fins didáticos e sem qualquer lógica de negócios real, a Listagem 1.14 deixa claro como funciona a injeção de dependência. *@Configuration* (linha 13) indica

que a classe *RefeicaoConfig* (linha 14) declara *beans* através de métodos para serem processadas pelo *container* do Spring e disponibilizadas em tempo de execução; ela prepara sua refeição. Para dar ênfase no desacoplamento de tipos, foi utilizada herança para o contexto deste exemplo, mas, para maior desacoplamento, poderia ser interface. Assim, *Refeicao* (linha 5) conhece apenas *Feijao* (linha 1), sem conhecer a especialidade dele que, nesse caso, é *FeijaoPreto* (linha 3). Poderia ser *FeijaoBranco*, outra especialidade de *Feijao* - depende do seu objetivo. E a classe que faz uso recebe a instância injetada de *Refeicao*.

Listagem 1.14. Código exemplo de injeção de dependência

```
1.  class Feijao {}
2.
3.  class FeijaoPreto extends Feijao {}
4.
5.  class Refeicao {
6.
7.      Feijao feijao;
8.
9.      Refeicao(Feijao feijao) {
10.         this.feijao = feijao;
11.     }
12. }
13. @Configuration
14. public class RefeicaoConfig {
15.
16.     @Bean
17.     Refeicao prepararRefeicao() {
18.         return new Refeicao(new FeijaoPreto());
19.     }
20. }
```

A classe *MinhaRefeicao*, apresentada na Listagem 1.15, é anotada com *@Component*, indicando que ela será detectada e inicializada através de *component scanning*, que será executado automaticamente ao iniciar uma aplicação Spring Boot. Isso permite o uso de injeção com *@Autowired*, sendo aqui uma injeção de construtor, onde é obtida uma instância de *Refeicao* criada pelo método *prepararRefeicao* na classe *RefeicaoConfig*. Assim, Spring permite desacoplamento entre objetos no desenvolvimento de aplicação e sua lógica de negócios. Inclusive entre seus módulos e a aplicação desenvolvida, como pôde ser visto na definição de *bean* e componente, e como será apresentado mais à frente na implementação de outros componentes.

Listagem 1.15. Código exemplo de injeção de dependência

```
1.  @Component
2.  public class MinhaRefeicao {
3.
4.      Refeicao refeicao;
5.
6.      @Autowired
7.      public MinhaRefeicao(Refeicao refeicao) {
8.          this.refeicao = refeicao;
9.      }
10. }
```

1.4.1.2. Spring *Aspect Oriented Programming*

Spring também oferece desacoplamento através de AOP, de acordo com [Wall 2015], que permite acesso a funcionalidades já desenvolvidas no seu programa por meio de componentes reutilizáveis. Ele permite definição dos seus próprios aspectos, bem como utilizar seus aspectos existentes, como no seu serviço de segurança *Spring Security*, que protege o sistema que o implementa realizando autorização antes de acessar suas funcionalidades. Pode ser utilizado também para gerenciamento de transações e *logging*.

1.4.1.3 Spring Web

O objeto desta seção é apresentar o desenvolvimento *Web* com o módulo Spring *Web* MVC, sendo *Movel-View-Controller*, de acordo com [Sommerville 2015], uma abordagem reconhecidamente utilizada para construção de sistemas, onde a visão do cliente da aplicação é desacoplada da sua lógica de negócios. Com esse módulo, é possível definir métodos que recebem requisições HTTP e devolvem respostas em JSON através de API *RESTful* ou uma página HTML. Também é utilizado uso também do módulo de acesso a dados Spring Data.

Spring roda na *Java Virtual Machine* (JVM) e possui duas arquiteturas de pilha: a tradicional síncrona e a mais recente nos últimos poucos anos, assíncrona. A arquitetura síncrona funciona com entrada e saída (E/S) bloqueante e atende uma requisição por *thread*. Ela é construída sobre a API *Servlet*, conhecida como *Servlet Stack,* dando origem ao módulo *Spring Web MVC,* que está presente no seu *framework* desde o início. A arquitetura assíncrona foi construída para aproveitar os processadores *multicore,* funcionar com E/S não bloqueante e atender muitas requisições concorrentes. Ela é conhecida por *Reactive Stack* e, em Spring, seu módulo é o *Spring WebFlux*. A arquitetura a ser utilizada depende do caso de uso e a escolha é sua.

1.4.1.4 Spring Data

Persistência de dados tende a gerar código *boilerplate* para criar a conexão ao banco de dados, realizar consulta, processar o resultado e finalizar a conexão, além de *queries* repetidas para cada tipo de consulta e tabela modelada. *Spring Data* abstrai o código *boilerplate* necessário para procedimentos como esses e ainda oferece métodos prontos para persistência quando implementados em conjunto com seus objetos. Isso evita erros lógicos de implementação de *query,* fechamento de recursos e tratamento de erros. Ele tem suporte a *Java Database Connectivity* (JDBC) e *Object Relational Mapping* (ORM)

com Hibernate e *Java Persistence API* (JPA). Funciona com bancos SQL e NoSQL. Ainda, possui suporte às transações com abstração para *Java Message Service* (JMS) para integração assíncrona com outras aplicações através de mensagens e faz uso de AOP para gerenciamento de transações.

1.4.1.5 Spring Test

Um processo fundamental na construção de sistemas é o *Test-Driven Development* (TDD), conforme Langr [2015], onde requisitos de software são tratados de forma específica. Testes podem ser feitos em funcionalidades pequenas e até em funcionalidades integradas, onde são tratadas as partes do sistema em conjunto. Spring permite testes com JUnit ou TestNG e também fornece suporte com seu módulo de teste para testar funcionalidades desenvolvidas com eles, inclusive em conjunto com injeção de dependência. Por exemplo, como requisições HTTP, configuração de segurança e persistência de dados. Ele disponibiliza objetos *mock* configuráveis*,* que são implementações de abstrações de contexto e ambiente de execução.

1.4.1.6. Configuração Inicial Spring Boot

Spring Boot é o ponto de partida para aplicações Spring. Ele foi projetado para facilitar a criação de aplicações prontas para produção onde você pode rapidamente iniciar o desenvolvimento com pouca configuração, incluindo integração com outras tecnologias. Spring faz uso de um modelo de programação baseado em anotações nas classes Java, permitindo substituição da programação baseada em XML, para o qual também possui suporte. Spring Boot levou isso para um outro nível dando suporte à autoconfiguração para códigos *boilerplates* de configuração e desenvolvimento.

A partir deste ponto é apresentado um fluxo de desenvolvimento com Spring Boot para entendê-lo na prática. Uma interface para criação do projeto inicial é disponibilizada como uma página *Web* em [Pivotal 2018b]. Através dela, é possível escolher a ferramenta para *build* e gerenciamento de dependências, a linguagem de desenvolvimento, o tipo de *packaging*, a versão do Spring Boot e as dependências necessárias. Aqui, é utilizado projeto Maven, linguagem Java com *packaging* Jar, que é o mais moderno, Java 8, Spring Boot 2.0.4 e as dependências *Web*, para Spring MVC e arquitetura *Servlet Stack,* JPA (do inglês, *Java Persistence API),* banco de dados H2 e *Thymeleaf*. É utilizado somente *Servlet Stack*. Por padrão, Spring utiliza o *servlet Apache Tomcat* embutido, mas pode ser configurado para outros *containers* compatíveis com a versão *servlet* mínima exigida. Após *download* do projeto, no diretório principal está o arquivo *pom.xml* com toda a configuração do projeto Maven. Em *src/main/java/nome/do/pacote* existe uma classe que inicia a aplicação Spring Boot no famoso método *main* Java com o método estático *run* da classe *SpringApplication*. Ela também possui uma anotação *@SpringBootApplication que* habilita autoconfiguração e busca por todas as *Spring beans* existentes no projeto ao ser executado.

Beans são instancias de objetos pertencentes ao *container Inversion of Control* (IoC) do Spring. IoC é um princípio da engenharia de software que também é conhecido como *Dependency Injection* (DI). A instância de um objeto é controlada pelo *container* sendo "injetada" em outro objeto que possui dependência com o primeiro. Isso aumenta a modularidade do programa e o torna mais extensível, como demonstra [Hall 2017]. *Beans* possuem escopo padrão *singleton* por *container*, o que significa que é criada apenas uma instância por *container*, semelhantemente ao padrão definido por [Gamma

et al 1996], podendo inclusive ser configurado com outras opções de escopo. Com isso, Spring permite um alto nível de desacoplamento entre objetos.

Antes de começar o desenvolvimento, é uma boa prática executar o projeto para garantir que está tudo funcionando. Pode-se fazer isso a partir de um *Integrated Development Environment* (IDE) com suporte ao gerenciador de projeto Apache Maven ou diretamente com o Maven a partir de um terminal. A partir de um IDE é simples construir e executar o projeto com sua interface. No terminal, pode-se utilizar os arquivos *mvnw*, escrito em *shell script*, para sistemas Unix e *mvnw.cmd,* escrito em *batch*, para *Windows*. Basta executar um dos seguintes comandos: *./mvnw spring-boot:run* ou *mvnw.cmd spring-boot:run*. Também é possível gerenciar o projeto diretamente pelo comando *mvn,* independente desses arquivos, com Maven configurado no sistema. Ao inserir o comando de execução, todas as classes são compiladas e os arquivos *.class* resultantes são armazenados no diretório *target* criado automaticamente.

1.4.1.7. RESTful Web Service com Spring MVC

Para o desenvolvimento do projeto é utilizado o módulo Spring MVC (do inglês, *Model View Controller*), que mapeia requisições com anotações. Até o presente momento existe foi obtido um projeto criado pelo inicializador do Spring, que vem pronto para execução. O endereço padrão de execução do servidor é *localhost:8080*, e pode ser acessado diretamente no navegador ou com outro cliente HTTP. Agora é criada uma API *REST* para simplesmente retornar uma mensagem. Ela pode ser visualizada em *localhost:8080/mensagens.*

Na Listagem 1.16 é possível observar um serviço REST que mapeia requisições HTTP do tipo GET no caminho */mensagens* (linha 2). Para entender melhor o funcionamento, é preciso conhecer os estereótipos *@Componente* e *@Controller* fornecidos pelo Spring. Anotações de estereótipos declaram componentes que serão identificados e registrados como *beans* gerenciadas pelo *container* IoC e iniciadas junto com a aplicação. *@Component* é um tipo genérico para componentes. *@Controller* é uma especialização de *Component* utilizada para representar controladores *Web* que recebem requisições. Ela costuma ser usada com *@RequestMapping* (linha 2) para mapear requisições em nível de classe ou de método. Em nível de método, normalmente utiliza-se especializações para identificar métodos HTTP como *@GetMapping* (linha 5), que possui a mesma semântica do código a seguir. E assim por diante para os métodos POST, PUT, DELETE E PATCH. *@RestController* (linha 1) é uma especialização de *Controller* com adição de *@ResponseBody,* que escreve diretamente no corpo da resposta HTTP.

Listagem 1.16. Código de serviço REST que mapeia requisições HTTP

```
1. @RestController
2. @RequestMapping("/mensagens")
3. class ControladorMensagem {
4.
5.     @GetMapping
6.     String encontrarMensagem() {
7.         return "Opa!";
8.     }
9. }
```

Classes com papel de controlador e seus métodos que recebem requisição são normalmente públicos, mas foram omitidos aqui. Agora, um objeto é retornado no formato *Javascript Object Notation* (JSON). Para tanto, um sistema de avaliação é simulado. Além disso, é definada uma classe *Avaliacao* na Listagem 1.17 em que seus métodos *gets* e seu construtor estão omitidos.

Listagem 1.17. Código de sistema de avaliação

```
1. class Avaliacao {
2.     int classificacao;
3.     String comentario;
4. }
```

Na Listagem 1.18 o controlador retorna um objeto *Avaliacao* (linha 7), no formato JSON, requerendo obrigatoriamente métodos *get* para os atributos da classe.

Listagem 1.18. Código da classe *ControladorAvaliacao*

```
1. @RestController
2. @RequestMapping("/avalicoes")
3. class ControladorAvaliacao {
4.
5.     @GetMapping
6.     Avaliacao encontrarAvaliacao() {
7.         return new Avaliacao(5, "Ótimo");
8.     }
9. }
```

Dando continuidade, o código da Listagem 1.19 apresenta como acessar recursos em *Web services* através de identificadores. Para isso, adiciona-se um parâmetro na API. Mas antes, é necessário incluir o novo campo *id* na classe *Avaliacao* e seu método *get*. O recurso pode ser acessado em *http://localhost:8080/avaliacoes/100*, onde 100 é um argumento para o ID do objeto procurado. Variáveis *Uniform Resource Identifier* (URI) podem ser declaradas entre {} e acessadas com *@PathVariable* no parâmetro do método (linha 2). Elas são convertidas automaticamente com suporte padrão a tipos como *long*, *double* e *String*.

Listagem 1.19. Código do método *encontrarAvaliacao*

```
1. @GetMapping("/{id}")
2. Avaliacao encontrarAvaliacao(@PathVariable long 1. id) {
3.     return new Avaliacao(id, 5, "Ótimo");
4. }
```

Até aqui, tem-se uma API que permite consulta, mas receber dados também é necessário e isso é demonstrado Listagem 1.20. Para receber requisições POST, basta utilizar *@PostMapping* e exigir um corpo na requisição com *@RequestBody* (linha 2) seguido do tipo do objeto esperado no parâmetro do método. Além disso, a classe *Avaliacao* precisa de um construtor padrão para instância do objeto. Construtor padrão não possui parâmetros e Java define um de forma implícita para cada classe. Se for definido algum outro, o padrão é sobrescrito e deve ser declarado de forma explícita, se desejado.

Listagem 1.20. Código do método *encontrarAvaliacao*

```
1. @PostMapping
2. void criarAvaliacao(@RequestBody Avaliacao avaliacao) {
3.
4. }
```

Para os métodos PUT ou PATCH, pode-se usar dois parâmetros: um identificador com *PathVariable* e *RequestBody* para o conteúdo que se pretende atualizar. Para o método DELETE, basta receber um identificador com *PathVariable* para procurar o recurso alvo. É exemplificado um JSON, observe Listagem 1.21, para a classe *Avaliacao*, enviado no corpo da requisição de métodos que exigem um, como o nosso método POST anterior. Assim, tem-se uma API *RESTful* que permite comunicação entre cliente e servidor com transmissão de dados no formato JSON.

Listagem 1.21. Exemplo de arquivo JSON utilizado na comunicação cliente/servidor

```
{
    "id": 100,
    "classificacao": 5,
    "comentario": "Ótimo!"
}
```

É possível utilizar o Spring MVC *Test* para testar a API ou um cliente HTTP como a aplicação *Postman*, um *API Development Environment* (ADE) gratuito.

1.4.1.8. Persistência de Dados com Spring Data JPA

Spring Data reduz significativamente a quantidade de código *boilerplate* necessária para implementar a camada de acesso a dados. O acesso da camada de dados será implementado com Spring *Data* JPA. Ele funciona com uma das mais famosas implementações da JPA: *Hibernate*. Spring Boot fornece o arquivo *application.properties* para configuração do projeto em *src/main/resources*. Por conveniência, pode-se usar um arquivo YAML, pois ele possui um formato de

configuração em hierarquia. Assim, o arquivo *application.yml* é criado no mesmo local e o outro é dispensado. O banco de dados utilizado será o H2, um banco relacional em memória que funciona apenas durante a execução da aplicação. Spring Boot fornece integração embutida para ele e não exige instalação, tornando-o conveniente para testes e para este tutorial. Para utilizar outro banco relacional, inclusive para persistência em memória não volátil, como MySQL, seria necessário apenas incluir seu conector em *pom.xml* e a configuração adequada em *application.yml*; o código Java da entidade permaneceria o mesmo.

A Listagem 1.22 mostra como acessar o banco de dados H2 por meio da configuração padrão do Spring *Boot* (linha 4). Neste código também é habilitada a visualização das *queries* (linha 6) realizadas pela JPA. Isso é feito adicionando-se o trecho de código da Listagem 1.22 no arquivo *application.yml.*

Listagem 1.22. Configuração do acesso ao banco de dados H2

```
1. spring:
2.   h2:
3.     console:
4.       enabled: true
5.   jpa:
6.     show-sql: true
```

A aplicação pode ser executada e o painel H2 visualizado em *localhost:8080/h2-console* em um navegador. São utilizados os mesmos dados no painel conforme Figura 1.6. Ao testar a conexão deve ser visualizada uma mensagem de sucesso.

Figure 1.6. Painel de acesso ao banco H2 num navegador com Spring, Fonte: Autores [2018]

Após configuração, segue a atividade de desenvolvimento que pode ser observada no código da listagem 1.23. JPA faz uso de anotações pertencentes ao pacote *javax.persistence* para mapear objetos para tabelas relacionais (ORM). A classe *Avaliacao* é atualizada e o restante dela pode permanecer o mesmo. *@Entity* (linha 1)

indica que *Avaliacao* é uma entidade JPA mapeada para uma tabela relacional com todos seus atributos como colunas. *@Id* (linha 4) especifica a chave primária da tabela e *@GeneratedValue* (linha 5) indica que o ID deve ser gerado automaticamente. Um construtor padrão é necessário porque JPA exige. A tabela e suas colunas podem ter seus nomes personalizados com as anotações *@Table(name = "nometabela")* no topo da classe e *@Column(name = "nomecoluna")* nos atributos. Caso não sejam especificados, são utilizados os mesmos nomes da classe e dos atributos. Existem outras personalizações normalmente utilizadas, como validação de dados dos atributos das classes com anotações contidas no pacote *javax.validation.constraints*.

Listagem 1.23. Código da Classe *Avaliacao*

```
1.  @Entity
2.  class Avaliacao {
3.
4.      @Id
5.      @GeneratedValue
6.      private long id;
7.
8.      public Avaliacao() {}
9.  }
```

Alguns tipos de operações em bancos de dados são comuns em muitos sistemas, como as operações CRUD, por exemplo. A característica mais destacada do Spring *Data* é a capacidade de criar repositórios de forma automática. Repositórios do Spring são interfaces que podem ser definidas para acessar dados. Eles disponibilizam métodos comuns para persistência e podem criar *queries* a partir do nome de métodos personalizados em tempo de execução. Para *queries* mais complexas, pode-se também defini-las utilizando *Java Persistence Query Language* (JPQL) ou SQL nativa com *@Query* acima do método. A Listagem 1.24 apresenta a classe que representa um repositório para a classe *Avaliacao*.

Listagem 1.24. Código do repositório da classe *Avaliação*

```
1. @Repository
2. interface RepositorioAvaliacao extends
3.     CrudRepository<Avaliacao, Long> {
4.
5. }
```

Apenas com o código da Listagem 1.24 já é possível realizar operações no banco. *@Repository* indica que uma classe possui papel *Data Access Object* (DAO). *CrudRepository* (linha 3 do código previamente mencionado), como o nome sugere, fornece funcionalidades CRUD para a entidade especificada no tipo genérico esperado, seguida pelo tipo da sua chave primária. Agora, o repositório é acessado no *ControladorAvaliacao* (Listagem 1.25) para aprimorar a API.

Listagem 1.25. Código da classe *ControladorAvaliacao*

```
1. class ControladorAvaliacao {
2.
3.     RepositorioAvaliacao repositorioAvaliacao;
4.
5.     @Autowired
6.     ControladorAvaliacao(RepositorioAvaliacao r) {
7.         this.repositorioAvaliacao = r;
8.     }
9. }
```

Discorrendo um pouco mais sobre o código da Listagem 1.25, observa-se que nele é definido um atributo do tipo *RepositorioAvaliacao* (linha 3) que é iniciado no construtor da classe. *@Autowired* (linha 5) solicita a instância de uma *bean* criada anteriormente, nesse caso, do *RepositorioAvaliacao*. O restante do código anterior da classe pode permanecer o mesmo. O *Autowired* pode ser removido do construtor e Spring vai reconhecer a solicitação da *bean* do mesmo modo. O principal objetivo de utilizá-la aqui foi mostrar o seu funcionamento. Com acesso à instância do repositório, serão utilizados os métodos de criar e encontrar avaliação (linha 2 e 8 da Listagem 1.26).

Listagem 1.26. Código dos métodos *criarAvaliacao* e *encontrarAvaliacao*

```
1.  @PostMapping
2.  void criarAvaliacao(@RequestBody Avaliacao avaliacao) {
3.      repositorioAvaliacao.save(avaliacao);
4.  }
5.
6.  @GetMapping("/{id}")
7.  Avaliacao encontrarAvaliacao(@PathVariable long id) {
8.      return repositorioAvaliacao.findById(id)
9.      .orElse(null);
10. }
```

A Listagem 1.27 apresenta trechos de código que ajudam a entender como tirar proveito ainda mais do Spring *Data*. Nela, uma *query* é criada a partir do nome de um método personalizado com palavras-chave fornecidas pelo Spring. Em *RepositorioAvaliacao*, é adicionado o método a seguir para encontrar todas as avaliações por valor de classificação. *find* indica o inicio de uma busca; *Classificacao* corresponde ao atributo da classe *Avaliacao*; e a palavra-chave *Equals* é utilizada para comparação. Em seguida, compare com a SQL nativa que tem a mesma semântica. Nela, *classificacao* corresponde à uma coluna da tabela de *Avaliacao* e *valor* corresponde ao parâmetro *int classificacao* do método *findByClassificacaoEquals*.

Listagem 1.27. Recursos do Spring *Data*

```
List<Avaliacao> findByClassificacaoEquals(int classificacao);

"select * from avaliacao where classificacao = valor;"
```

A nova busca é adicionada no *ControladorAvaliacao* com o método a da Listagem 1.28. Ele exige um parâmetro para filtrar avaliações (linha 1) de acordo com o valor de classificação. *@RequestParam* (linha 3) é utilizada para parâmetros de busca ou dados de formulário. Essa funcionalidade pode ser acessada através de uma requisição GET em *localhost:8080/avaliacoes?classificacao=5*.

Listagem 1.28. Método *encontrarPorClassificacao*

```
1. @GetMapping(params = "classificacao")
2. Iterable<Avaliacao> encontrarPorClassificacao(
3. @RequestParam int classificacao) {
5.     return repositorioAvaliacao
6.     .findByClassificacaoEquals(classificacao);
7. }
```

É possível também buscar avaliações por palavras contidas na *string* comentário. O método apresentado na Listagem 1.29 em *RepositorioAvaliacao*. Nele, as palavras-chave utilizadas são: *Containing* para encontrar avaliações que possuam uma *substring* no seu comentário e *IgnoreCase* para não diferenciar letras maiúsculas e minúsculas.

Listagem 1.29. Método *findByComentarioContainingIgnoreCase*

```
List<Avaliacao> findByComentarioContainingIgnoreCase(String
comentario);
```

Com isso, tem-se um RESTful *Web Service* integrado com persistência de dados. As *queries* realizadas podem ser acompanhadas no *console* e seus dados verificados no banco através do painel *Web* do H2 em *localhost:8080/h2-console*.

1.4.1.9. Lógica de Negócios como Serviço

Spring disponibiliza também um estereótipo *@Service* que é especialização de *@Component*. Ele é baseado no *Domain-Driven Design*, proposto por Evans [2003], onde podem ser oferecidas operações isoladas através de interfaces. Semelhantemente, pode também indicar uma fachada para a lógica de negócios. *@Service* é um estereótipo de propósito geral e cabe ao desenvolvedor definir bem a sua semântica. Nesse projeto, podemos abstrair o acesso aos dados como regra de negócios. Nossos controladores não vão conhecer o repositório. E, caso exista algum processamento antes ou depois da consulta no repositório, pode ser realizado no componente de serviço e nossos controladores - ou outras classes clientes - não precisam implementar nem conhecer seus detalhes.

Duas classes de serviço são adicionadas conforme o código da Listagem 1.30, sendo obtida uma instância do repositório na implementação do serviço através de inicialização no construtor, como anteriormente. No *ControladorAvaliacao* (linha 19), é obtida uma instância de *ServicoAvaliacaoImpl* (linha 6) através da interface *ServicoAvaliacao* (linha 1). Spring reconhece automaticamente a classe que implementa a interface pelo tipo de dado, desde que ela tenha anotação de componente, nesse caso, com *@Service* (linha 5). Todos os acessos anteriores ao repositório apresentados aqui também podem ser substituídos por acessos ao serviço.

Listagem 1.30. Classes de serviço

```
1. interface ServicoAvaliacao {
2.
3.     Optional<Avaliacao> encontrarPorId(long id);
4. }
5.  @Service
6.  class ServicoAvaliacaoImpl implements ServicoAvaliacao {
7.
8.      RepositorioAvaliacao repositorioAvaliacao;
9.
10.      ServicoAvaliacaoImpl(RepositorioAvaliacao r) {
11.          this.repositorioAvaliacao = r;
12.      }
13.
14.      @Override
15.      public Optional<Avaliacao> encontrarPorId(long id) {
16.          return repositorioAvaliacao.findById(id);
17.      }
18.  }
19.  class ControladorAvaliacao {
20.
21.      ServicoAvaliacao servicoAvaliacao;
22.
23.      ControladorAvaliacao(ServicoAvaliacao sa) {
24.          this.servicoAvaliacao = sa;
25.      }
26.  }
```

1.4.1.10. Página Web com Spring MVC

Spring permite a criação de páginas *Web* no *server-side* com diferentes tecnologias. Uma delas é *Thymeleaf*, um modelo Java que funciona com HTML5 e possui autoconfiguração com Spring Boot. Será criada uma simples página *Web* para demonstrar basicamente sua funcionalidade. Com configuração padrão, os arquivos são procurados em *src/main/resources/templates*. Lá, é criado um arquivo *inicial.html* e incluído o código a da Listagem 1.31. Nele, é feita a importação do *Thymeleaf* (linha 2), o que permite a utilização da palavra-chave *th* para acessar dados recebidos do *server-side* como tipos primitivos ou não primitivos, como *mensagem*, definida entre ${} (linha 3) para indicar uma variável.

Listagem 1.31. Páginas *Web* com *Thymeleaf*

```
1. <!DOCTYPE html>
2. <html xmlns:th="http://www.thymeleaf.org">
3. <h1 th:text="${mensagem}"></h1>
4. </html>
```

Agora, é preciso tratar requisições para esta página. Como pode ser observado na Listagem 1.32, a classe *ControladorInicial.java* é criada em *src/main/java/nome/do/pacote* com o código a seguir. *@Controller* (linha 1) indica que essa classe é um componente que trata requisições, de forma semelhante ao que vimos antes, mas não implementa *@ResponseBody* como em *@RestController*. O método *inicial* (linha 6) recebe um *Model* e retorna uma *view*, nesse caso, *inicial.html* (linha 8). Através do *model*, é possível mapear dados entre controlador e visão com chave-valor.

Listagem 1.32. Classes *ControladorInicial*

```
1. @Controller
2. @RequestMapping("/inicial")
3. class ControladorInicial {
4.
5.     @GetMapping
6.     String inicial(Model model) {
7.         model.addAttribute("mensagem", "Avaliações");
8.         return "inicial";
9.     }
```

Requisições e seus parâmetros podem ser tratadas com anotações como visto antes em *RestController*. Para exibir dados de um objeto, os trechos de código da Listagem 1.33 são inseridos nos arquivos HTML e Java. Para mais funcionalidades, a documentação *Thymeleaf* está disponível em *https://www.thymeleaf.org/*

Listagem 1.33. Códigos HTML e Java para exibir dados de um objeto

```
1.  <div th:object="${avaliacao}">
2.      <td th:text="${avaliacao.classificacao}"></td>
3.      <td th:text="${avaliacao.comentario}"></td>
4.  </div>
5.  @GetMapping
6.  String inicial(Model model) {
7.
8.      Avaliacao avaliacao = new Avaliacao(5, "Muito bom");
9.      model.addAttribute("avaliacao", avaliacao);
10.  }
```

O código fonte aqui apresentado está disponível no *Github*[18] em. Spring disponibiliza uma rica documentação em sua página oficial *https://spring.io/* onde é possível conhecer melhor os projetos fornecidos.

[18] https://github.com/rkulesza/webdev

1.5. Arquiteturas de Sistemas Web

As arquiteturas de sistemas Web evoluíram drasticamente desde a criação da Internet. No início os sistemas eram feitos utilizando a arquitetura CGI (do inglês, *Common Gateway Interface*). Esse recurso deu um grande poder aos servidores, já que passou a oferecer a capacidade de executar scripts de códigos – geralmente *Perl* – ao processar as requisições HTTP, fazendo com que os sistemas Web pudessem processar requisições de uma forma mais dinâmica [Hunter e Crawford 2001].

Em seguida outro problema no início da Web: era muito difícil separar os códigos de apresentação e lógica das aplicações, dificultando o seu desenvolvimento. Surgiram então os *template systems*, eles permitiam que códigos executáveis de uma linguagem de programação fossem inseridos diretamente nos arquivos responsáveis pela apresentação do sistema. Assim, dividia-se melhor as 2 camadas (apresentação e lógica) [Fields e Marc 2012]. Depois disso, diversas arquiteturas surgiram, entre elas o "modelo 2" da arquitetura MVC, que mais tarde tornou-se um dos principais modelos de sistemas Web [Sommerville 2015] e impulsou tecnologias como os frameworks Struts, Tapestry e JSF (do inglês, *Java Server Pages*). Também nessa época foram desenvolvidos frameworks para facilitar o mapeamento entre modelos orientado à objetos e relacionais (por exemplo, o *Hibernate*), dando origem as arquiteturas em 3 camadas (apresentação, lógica de negócio e dados) [Fields e Marc 2012].

Com o crescimento do uso dos sistemas em ambiente corporativo e de acesso global, foi necessário a divisão do processamento de 3 para n camadas [Fowler et al. 2002], dando origem às plataformas de execução distribuída como o JEE (do inglês, *Java Platform, Enterprise Edition*), Net. e Spring. Surgiram também protocolos de comunicação (SOAP, REST, etc.) que permitiam que os sistemas se comunicassem independentemente da linguagem de programação utilizada e facilitando a integração de sistemas heterogêneos e legados. Com isso, os desenvolvedores não estavam mais somente construindo aplicações que serviam conteúdos para os navegadores; mas sim, sistemas complexos que envolviam várias camadas de comunicação interna e externa (com outros sistemas) [Holdner 2018]. A partir daí os sistemas cresceram bastante, e a quantidade de usuários aumentou consideravelmente, fazendo com que esses sistemas se tornassem grandes demais, transformando-os em gigantes sistemas monolíticos [Newman 2015]. Esses sistemas possuem diversos problemas de escalabilidade e desempenho quando muitos usuários o utilizam. A solução foi encontrada em arquiteturas menos monolíticas e mais distribuídas – como as de SOA (do inglês, *service-oriented architecture*) utilizando o conceito de Microserviços e persistência poliglota [Sadalage e Fowler 2012]. Esses modelos possuem uma melhor distribuição de cada serviço do sistema, fazendo com que a carga de requisições em cada um seja melhor distribuída, melhorando consideravelmente requisitos de escalabilidade (por exemplo, balanceamento de carga e alta confiabilidade) [Newman 2015].

Como já citado anteriormente, grandes avanços também foram feitos na camada de apresentação no lado do cliente, diversos frameworks foram lançados e permitem que os sistemas Web tenham uma desempenho e usabilidade comparáveis com sistemas desktop tradicionais [Scott 2016]. Esses frameworks utilizam arquiteturas SPA [Scott 2016], atualizando somente o necessário através do uso de versões mais recentes do *Javascript* (por exemplo, ECMAScript versão 5 e 6) e comunicações AJAX com o

servidor [Scott 2016]. Esse modelo remove a responsabilidade de gerar a camada de visão dos servidores, deixando os sistemas mais leves, rápidos e fluídos [Scott 2016].

Nesse contexto e reforçado pelo mapa de opções que podemos observar na Figura 1.7, existem atualmente inúmeras opções de plataformas de desenvolvimento (linguagens, APIS, bibliotecas, frameworks etc.) para sistemas, Na próxima seção é descrito um estudo de caso que demonstra um conjunto de opções do estado da técnica em relação ao uso de tecnologias (cliente e servidor) e a aplicação de conceitos de arquiteturas modernas para sistemas Web.

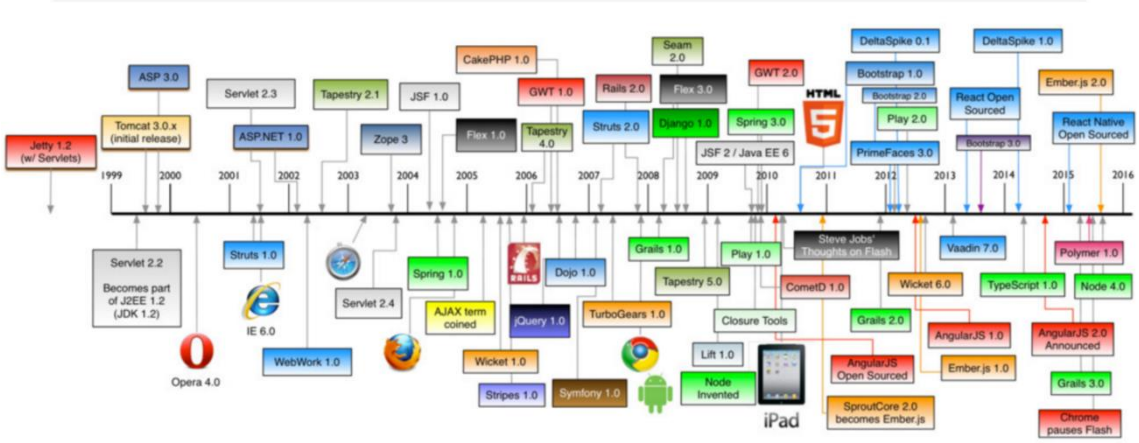

Figura 1.7 – Plataformas para desenvolvimento de sistemas Web. Fonte: Raible [2018].

1.5.1. Estudo de Caso: Você Digital

Você Digital compreende um projeto de pesquisa e desenvolvimento entre o LAVID/UFPB e o TCE-PB para modelagem e desenvolvimento de uma plataforma computacional colaborativa de governo eletrônico. O principal objetivo é permitir a automação de diagnósticos e escutas populares, que possibilite uma melhor interação e comunicação entre a sociedade e os gestores públicos.

Adicionalmente, a solução pretende explorar a inteligência coletiva presente nas redes, construindo uma participação cidadã, o que pode ajudar tanto na redução de custos no processo de geração destas diretrizes, bem como na promoção da transparência e confiabilidade com um real e efetivo ganho de tempo diante da visualização virtual e democrática de demandas e necessidades da sociedade. Como ferramenta de gestão pública digital a ideia é avaliar parte dos serviços públicos como também fomentar a participação popular no processo de tomada de decisões de auditores e gestores públicos. Como forma de avaliar a plataforma, no projeto está em desenvolvimento um aplicativo que utilizará novos métodos capazes de aumentar o envolvimento de cidadãos no contexto de diagnóstico de problemas em diversas áreas de gestão pública (por exemplo, educação, saúde e segurança).

1.5.1. 1. Projeto arquitetural de alto nível

Na Figura 1.8 é possível visualizar o projeto arquitetural de alto nível do sistema “Você Digital” com seus subsistemas destacados na cor azul. Os possíveis sistemas internos do TCE-PB estão destacados em laranja e os sistemas externos são visualizados no canto

superior da ilustração (ver na Figura 1.8 “Sistemas com *OpenID*” e “*Google Maps* e *Google Places*”). Tanto uma parte do sistema “Você Digital”, como os sistemas internos do TCE-PB utilizarão a infraestrutura de *datacenter* e virtualização disponível atualmente no TCE-PB.

O sistema "Você Digital" é composto por dois grandes subsistemas: 1) 2 (dois) sistemas de software cliente (*front-end*) e 2) 1 (um) sistema de retaguarda (*backend*). O primeiro (1) conjunto possui um aplicativo para dispositivos móveis compatíveis com a tecnologia *React Native* (ver figura 1.8 “App Você Digital”) e estará disponível para *download* nas lojas virtuais da *Apple* e *Google*. Além disso, também há um sistema de software cliente (ver na Figura 1.8, “Administração Você Digital”) que permitirá a administração do sistema “Você Digital” por meio de tarefas como gerenciamento de cadastros e dados, permissões de usuários, geração de relatórios estatísticos etc. Tal aplicativo é baseado na abordagem SPA (do inglês, *Single Page Application*) e na tecnologia *React*, de modo a permitir sua execução em qualquer navegador *Web* (*desktop* ou dispositivo móvel). O segundo conjunto (2) tem como papel o processamento dos cadastros de dados disponível no sistema, bem como processamento desses dados para gerar informações para os usuários. Este subsistema é dividido em três partes: I) Controlador: responsável pela distribuição de carga, alta disponibilidade e acesso seguro de dados e informações disponíveis para os aplicativos citados por meio de uma API *RESTful*; II) Servidores de aplicação (*containers* baseados na plataforma *Docker* e em tecnologias para desenvolvimento de arquiteturas de microserviços) responsável pelo tratamento da integração e processamento de dados interno e externos e particionamento das funcionalidades disponíveis para os softwares clientes e; III) Sistemas de Banco de Dados: responsável pelo armazenando dos dados utilizando persistência poliglota (neste módulo são utilizados tecnologias SQL e/ou NoSQL). A comunicação entre os aplicativos (1) e os servidores (2) é realizada por meio do protocolo HTTPS e APIs *RESTFul*.

Em relação a requisitos de integração do sistema “Você Digital” com os sistemas externos, foram utilizadas APIs disponíveis em sistemas compatíveis com *OpenID* para permitir autenticação externa (por exemplo, redes sociais) de modo a não ser necessário um cadastro no sistema “Você Digital” para ter acesso aos serviços do aplicativos. Do mesmo modo, foram utilizadas as APIs da plataforma *Google Maps* de modo a obter informações de geolocalização (*Google Maps*) e pontos de interesses geolocalizados cadastrados por pessoas físicas e jurídicas (*Google Places*).

Já para a integração com os sistemas internos, foi realizado um mapeamento de que dados (que podem estar um banco de dados SQL ou servidores) e/ou informações que seriam necessárias. A partir dessa identificação, a equipe do TCE-PB ofereceu uma API *RESTFul* para comunicação entre os sistemas. De forma análoga, o sistema “Você Digital”, oferece APIS *RESTFul* para o TCE-PB acessar dados.

De acordo com a Figura 1.8, o subsistema servidor de aplicação foi organizando nos seguintes módulos: AAA (do inglês, *Authentication, Authorization and Accounting*): trata-se dos procedimentos relacionados à autenticação, autorização e auditoria. Como se sabe, a autenticação verifica a identidade dos usuários, a autorização lida com as permissões, ou seja, garante que um usuário autenticado somente tenha acesso aos recursos autorizados para seu perfil e, por fim, a auditoria está relacionada à ação de coleta de dados sobre o comportamento dos usuários em relação ao sistema.

Vale salientar que este módulo se comunica com serviços externos de autenticação e é o responsável por gerenciar as seções de modo assíncrono e utilizando um banco de dados que não utilize sistema de arquivos baseado em disco; Administração (CRUD): gerencia todas as entidades do modelo de classes, ou seja, é responsável por realizar as quatro operações básicas de criação, consulta, atualização e destruição em banco de dados; Publicação (*inputs*): este módulo é a principal fonte de entrada de dados do sistema "Você Digital", sendo responsável por receber todas as informações geradas pelos usuários como avaliações, comentários, gravações de vídeos e fotos. Além desta função, este módulo também acumula a responsabilidade de realizar o tratamento de segurança dos dados. Em virtude da natureza do sistema, faz-se necessária a aplicação de filtros de textos nos comentários a fim de identificar colocações inadequadas, bem como também é adequado "sanitizar" os dados para evitar ataques de injeção. Outra funcionalidade é prover autenticação do aplicativo a fim de evitar fraudes por meio de programas de inteligência artificial (robôs) que podem ser usados para manipulação de informação; Busca: lida com pesquisas de baixa granularidade, como consultas diversas ao banco de dados; e por fim, Consumo (*outputs*): este módulo utiliza o módulo de Busca para realizar *Data Analytics*, ou seja, gerar dados estatísticos, transformação de dados, gráficos, relatórios, dentre outras análises.

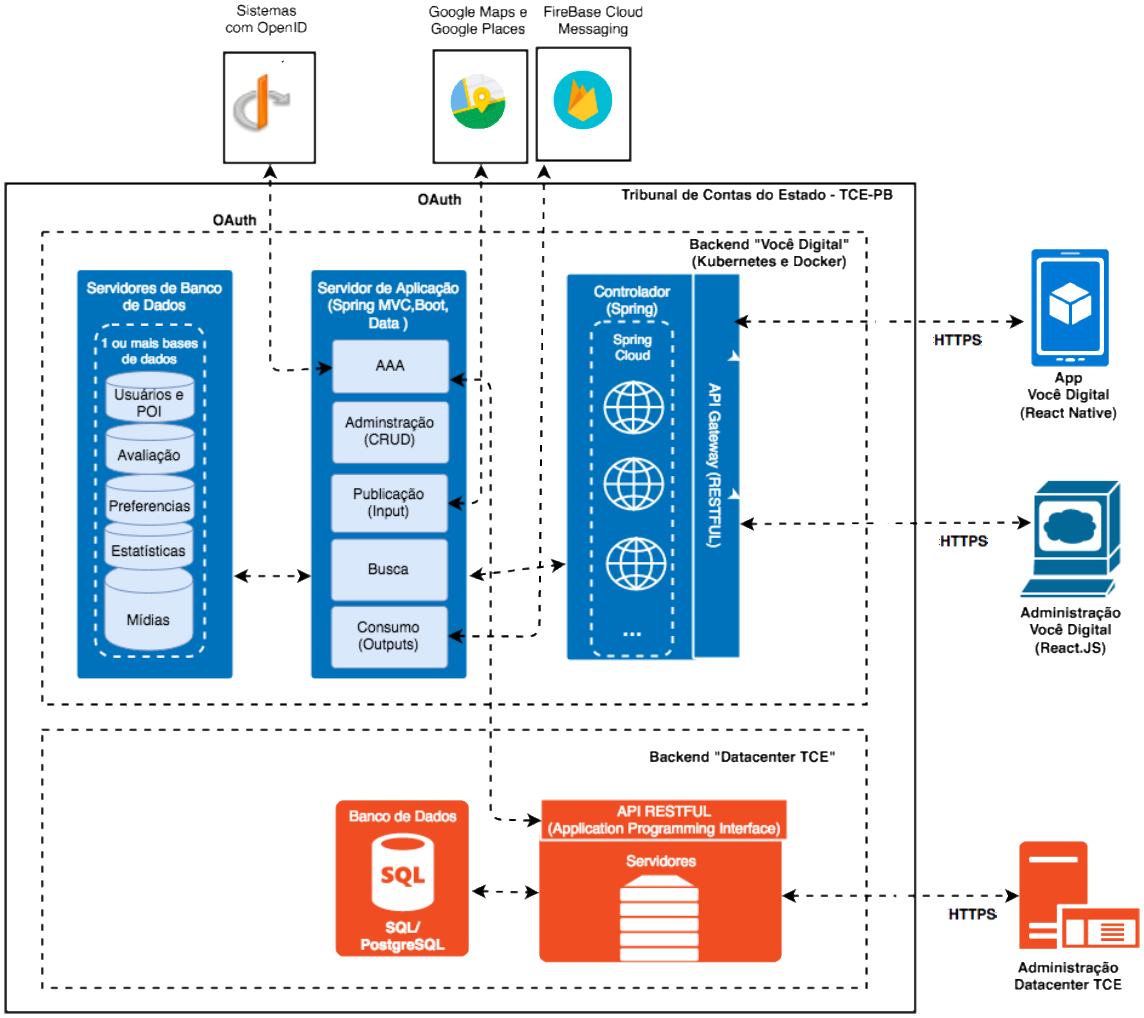

Figura 1.8. Projeto arquitetural de alto nível do sistema "Você Digital", Fonte: Autor [2018]

Dando continuidade ao detalhamento da arquitetura, o subsistema servidor de Banco de Dados foi organizando nas seguintes bases: Usuários e POI: essa base guarda as informações de cadastro de usuários e os seus pontos de interesse; Avaliação: essa base guarda as informações relacionadas ao histórico das avaliações realizadas pelos usuários; Preferências (perfil): essa base guarda informações dinâmicas relacionadas aos usuários, como: IP, Latitude e Longitude, dentre outras; Estatísticas: essa base guarda estatísticas persistentes quem podem ser utilizadas pelo módulo Consumo do subsistema servidor de aplicação; e por último, Mídias: que é uma base que guarda todas as mídias geradas pelos usuários, como textos, imagens, vídeos e áudios.

Do ponto de vista de tecnologia, após estudos realizados, foi definida a adoção das soluções do ecossistema *Spring* para a implementação do subsistema Servidor de Aplicações. Em resumo, o *framework Spring* é ferramenta utilizada para aumentar a produtividade na escrita de aplicações corporativas explorando conceitos como injeção de dependência e inversão de controle. Além da tecnologia *Spring* no desenvolvimento da lógica de negócio e acesso aos dados no servidor, também será adotado no subsistema Controlador soluções da suíte *Spring Cloud*[19], que traz funcionalidades para realizar a configuração, roteamento, distribuição de carga e alta disponibilidade para os serviços implementados.

1.6. Conclusão

Este capítulo apresentou um breve histórico e tecnologias atuais de plataformas de desenvolvimento de software *Web* do lado do cliente e do servidor. Foi descrito o histórico da evolução de modelos arquiteturais de Sistemas *Web* e também apresentado um estudo de caso por meio das soluções que foram desenvolvidas no Tribunal de Contas do Estado da Paraíba (TCE-PB) em parceira com o Laboratório de Aplicações de Vídeo Digital (LAVID) da Universidade Federal da Paraíba (UFPB). Tal solução adotou as tecnologias e modelos arquiteturais discutidos em um projeto real. A principal contribuição deste trabalho foi disseminar o histórico dos sistemas *Web* e elucidar as tecnologias e arquiteturas utilizadas hoje e tendências para o futuro.

References

[Benioff 2018] Benioff, M., “Welcome to Web 3.0: Now Your Other Computer is a Data Center | TechCrunch”, Disponível em: <http://techcrunch.com/2008/08/01/welcome-to-web-30- now-your-other-computer-is-a-data-center-2/>. Acesso em: 26/09/2018.

[Berners-lee 1996] Berners-lee, T. (1996) “WWW: past, present, and future”. Computer, v. 29, n. 10, p. 69–77.

[Bonér 2016] Bonér, J., (2016) “Reactive Microservices Architecture”, Sebastopol: Pearson Education, Inc,

[Burégio 2014] Burégio, V. A. A. (2014) “Social machines: a unified paradigm to describe, design and implement emerging social systems”, Doctor of Computer Science (PhD): Computer Science, Federal University of Pernambuco, Recife, Brasil.

[19] Projeto Spring Cloud. Disponível em: https://cloud.spring.io

[Burns 2018], Burns B., “Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services”, O'Reilly Media, Inc., 2018.

[Burns 2013] Chedeau, C. (2013) “React’s diff algorithm”. Disponível em: https://calendar.perfplanet.com/2013/diff, Acesso em: 01/8/2018.

[Deitel et al. 2010] Deitel, P. J. et al., “Internet & World Wide Web”, Boston: Pearson Education, Inc, 2012.

[Facebook 2018] Facebook (2018) “React - a javascript library for building user interfaces”. Disponível em: https://reactjs.org, Acesso em: 01/8/2018.

[Fields e Marc 2012] Fields, D. K., Mark A. (2002) “Web development with JSP. Greenwich”, Manning.

[Fox e Hao 2018] Fox, R. e Hao, W., “Internet Infrastructure”, New York: Taylor & Francis Group, LLC, 2018.

[Fowler et al. 2002] FOWLER, M. et al. (2002) “Patterns of Enterprise Application Architecture”, Addison Wesley.

[Gamma et al. 1996] Gamma, E., Helm, R., Johnson, R., Vlissides, J., (1996) “Design Patterns”, Boston: Pearson Education Corporate Sales Division.

[Garrett 2005] Garrett, J. J. (2005), “AJAX: A new approach to web applications | adaptive path”, Disponível em: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. Acesso em: 08/9/2018.

[Groef 2016] Groef, W. (2016) “Client- and Server-Side Security Technologies for JavaScript Web Applications”, Doctor of Engineering Science (PhD): Computer Science, Faculty of Engineering Science, Ku Leuven, Leuven.

[Hall 2017] Hall, G., “Adaptive Code”, Redmond: Online Training Solutions, Inc, 2017.

[Holdener 2008] Holdener T. (2008), “AJAX The definitive guide”. 1. ed.

[Hunter e Crawford 2001] Hunter, J. Crawford (2001), W. “Java Servlet Programming”. 2. ed.

[Java 2008] Java Servlet Specification (2018), Disponível em: https://javaee.github.io/servlet-spec/. Acesso em: 18/09/2018.

[Langr 2015] Langr, J., “Pragmatic Unit Testing in Java 8 with JUnit”, Raleigh: The Pragmatic Bookshelf, 2015.

[Mardan 2017] Mardan, A. “React Quickly: Painless web apps with React, JSX, Redux, and GraphQL.”, 2017

[Maximilien, Ranabahu e Gomadam 2008] Maximilien, E. M.; Ranabahu, A.; Gomadam, K., (2008) “An Online Platform for Web APIs and Service Mashups”, IEEE Internet Computing, v. 12, n. 5, p. 32–43.

[Mikowski e Powell 2013] Mikowski, M, e Powell, J, (2013) “Single page web applications: JavaScript end-to-end”. Manning Publications Co.

[Newman 2015] (2015) Newman, S., “Building Microservices”, Sebastopol: Pearson Education, Inc.

[Patterson e Fox 2012] Patterson, D., Fox, A. (2012) “Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing”, Strawberry Canyon LLC.

[Pivotal 2018a] Pivotal (2018), “Spring Framework”, Disponível em: https://spring.io/. Acesso em: 18/8/2018.

[Pivotal 2018b] Pivotal (2018), “Spring Initializr”, Disponível em: https://start.spring.io/. Acesso em: 18/8/2018.

[Postman 2018] Postman (2018), “Postdot Technologies”, Inc. Disponível em: https://www.getpostman.com/. Acesso em: 18/8/2018.

[Raible 2015] Raible, M. (2015) “Comparing Hot JavaScript Frameworks: AngularJS, Ember.js and React.js”, Available at: http://raibledesigns.com, 2015. Acesso em: 10/07/2018

[Raible 2018] Raible, M. (2018) “Front End Development for Back End Developers”, Available at: http://raibledesigns.com.

[RFC 7231 2018] RFC 7231 (2018) “Hypertext transfer protocol (http/1.1): Semantics and content”, Disponível em: https://tools.ietf.org/html/rfc7231, Acesso em: 26/09/2018.

[Sadalage e Fowler 2012] Sadalage, P. J.; Fowler, M. Nosql Distilled: A Brief Guide To The Emerging World Of Polyglot Persistence, 1. Ed.,Addison Wesley, 2013.

[Scott 2016] Scott JR, E. A. “SPA Design and Architecture Understanding single-page web applications. Manning Publications”. 1. Ed. 2016.

[Sommerville 2015] Sommerville, I., “Software Engineering”, 10. ed. London: Pearson Education, Inc, 2015.

[Thymeleaf 2018] Thymeleaf Template (2018), “The Thymeleaf Team”, Disponível em: https://www.thymeleaf.org/. Acesso em: 18/8/2018.

[Wall 2015] Wall, C. “Spring In Action”, 2. ed. Shelter Island: Manning Publication Co, 2015.

[Webber, Parastatidis e Robinson 2010] Webber, J., Parastatidis S., Robinson I. (2010) “REST in Practice: Hypermedia and Systems Architecture”, O'Reilly Media, Inc.

[Yu et al. 2008] Yu, J. et al., (2008) “Understanding Mashup Development”, IEEE Internet Computing, v. 12, n. 5, p. 44–52, set.